**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validation of input is important, and you should validate from all untrusted data sources. Proper input validation can help to eliminate most software vulnerabilities. It is important to be aware and suspicious of the external sources line command line arguments, network interfaces, environment variables and user-controlled files. |
| 1. Heed Compiler Warnings | You should always compile your code using the highest warning level available to you and to eliminate the warnings when issues are found. Use static and dynamic analysis tools to detect and eliminate any additional security flaws found within your code. |
| 1. Architect and Design for Security Policies | Creating a software architecture and design to enforce security policies in your code. When needing different privileges for sections of code it is good practice to divide these areas into subsections that are only using the privileges that they need at that time. |
| 1. Keep It Simple | Complex design increases the chance of having errors and vulnerability. Keep your design small and simple and enough to get the task at hand completed. The effort required to keep code secured drastically increases as the design becomes more complex. |
| 1. Default Deny | Keep access denied as a starting default, and only when certain requirements are met can access be granted. Do not start your code with access being granted because it will be easily exploited. |
| 1. Adhere to the Principle of Least Privilege | Each of the processes in the code should have a set of privileges needed to even access certain portions. If these small necessities are not met, then the code should not be able to be accessed. Keeping access denied until a set of privileges are met is the safest way to code. |
| 1. Sanitize Data Sent to Other Systems | Cleaning up the commands and data passed into the subsystem will help keep code secure, this helps to practice when working with command shells, relational databases and commercial off the shelf components. This is common to protect against SQL injections and similar hacking methods because hackers can invoke unused functionality of some components in order to gain unauthorized access. |
| 1. Practice Defense in Depth | Practice making multiple layers of defense in case the initial layer can be broken. Having multiple layers of defense is the best practice when trying to secure data or code. Having multiple defenses can make an exploit in the initial layer of defense unusable in the next layer of defense. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques can be used to identify and eliminate vulnerabilities. It is good to understand how certain methods and practices can be attacked and exploited. Fuzz testing, penetration testing and source code audits can be incorporated as effective quality assurance techniques. |
| 1. Adopt a Secure Coding Standard | Adopt or develop your own secure coding standards for your own development platform, making sure to cover all bases when writing your code to make it the most secure possible. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-DCL50-CPP] | Declarations and Initialization |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses a C-style variadic function to add a series of integers together. The function reads arguments until the value 0 is found. Calling this function without passing the value 0 as an argument result in undefined behavior. Furthermore, passing any type other than an int also results in undefined behavior. |
| #include <cstdarg>    **int** add(**int** first, **int** second, ...) {  **int** r = first + second;  **va\_list** va;  **va\_start**(va, second);  **while** (**int** v = **va\_arg**(va, **int**)) {  r += v;  }  **va\_end**(va);  **return** r;  } |
|  |

| **Compliant Code** |
| --- |
| In this compliant solution, a variadic function using a function parameter pack is used to implement the add() function, allowing identical behavior for call sites. Unlike the C-style variadic function used in the noncompliant code example, this compliant solution does not result in undefined behavior if the list of parameters is not terminated with 0. Additionally, if any of the values passed to the function are not integers, the code is ill-formed rather than producing undefined behavior. |
| #include <type\_traits>    **template** <**typename** Arg, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg f, Arg s) { **return** f + s; }    **template** <**typename** Arg, **typename**... Ts, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg f, Ts... rest) {  **return** f + add(rest...);  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 7.0p0 | LANG.STRUCT.ELLIPSIS | Ellipsis |
| Clang | 3.9 | Cert-dcl50-cpp | Checked by clang-tidy |
| Parasoft C\C++Test | 2022.1 | CERT\_CPP-DCL50-a | Functions shall not be defined with a variable number of arguments |
| Polyspace Bug Finder | R2022a | CERT C++ : DCL50-CPP | Checks for function definition with ellipsis notation |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-DCL41-C] | Any variables declared inside of a switch statement will only be useful within that switch block. These variables do not get initialized, which makes then unusable or not visible by the rest of the code in the program. |

| **Noncompliant Code** |
| --- |
| This code declares variables before the first case label and within the switch statement. This is poor because it will only use those values within that switch statement area. |
| |  | | --- | | #include <stdio.h>    **extern** **void** f(**int** i);    **void** func(**int** expr) {  **switch** (expr) {  **int** i = 4;  f(i);  **case** 0:  i = 17;  /\* Falls through into default code \*/  **default**:  **printf**("%d\n", i);  }  } | |

| **Compliant Code** |
| --- |
| The way to do this is to declare the statements and variables before the switch statement like shown below which makes the code run smoothly and be able to use that variable almost globally. |
| |  | | --- | | #include <stdio.h>    **extern** **void** f(**int** i);    **int** func(**int** expr) {  /\*  \* Move the code outside the switch block; now the statements  \* will get executed.  \*/  **int** i = 4;  f(i);    **switch** (expr) {  **case** 0:  i = 17;  /\* Falls through into default code \*/  **default**:  **printf**("%d\n", i);  }  **return** 0;  } | |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikey | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 7.0p0 | LANG.STRUCT.SW.BAD | Malformed Switch Statement |
| Clang | 3.9 | -Wsometimes-unitialized |  |
| Helix QAC | 2022.2 | C2008, C2882, C3234 |  |
| LDRA Tool Suite | 9.7.1 | 385 S | Fully Implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-STR50-CPP] | Copying or putting data into a buffer that is not big enough for the data will result in buffer overflow which is a very bad practice in coding C++ |

| **Noncompliant Code** |
| --- |
| This code sets a size for the char which creates a buffer with a size. If data is added into this that is bigger than the designated size it can couse serious issues in the code. |
| |  | | --- | | #include <iostream>    **void** f() {  **char** bufOne[12];  **char** bufTwo[12];  std::cin.width(12);  std::cin >> bufOne;  std::cin >> bufTwo;  } | |

| **Compliant Code** |
| --- |
| This initializes the variable as an unbounded string which can be initialized at any size and not cause any harm to the system. |
| |  | | --- | | #include <iostream>  #include <string>    **void** f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } | |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 7.0p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null termination  Buffer overrun  Type overrun |
| Helix QAC | 2022.2 | C++2835, C++2836, C++2839, C++5216 |  |
| LDRA | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially Implemented |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | Avoid overflow due to not zero terminated string  Avoid overflow when writing to buffer  Prevent buffer overflows from tainted data  Do not use ‘char’ buffer to store input from ‘std::cin’ |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-EXP53-CPP] | Memory needs to be allocated correctly in order to be used in the right way and to be fully initialized. Without allocation the right amount of memory or the memory in the right way the code will not function as desired. |

| **Noncompliant Code** |
| --- |
| Memory is allocated here but the memory is not initialized and this will break the program. |
| |  | | --- | | #include <iostream>    **void** f() {  **int** \*i = **new** **int**;  std::cout << i << ", " << \*i;  } | |

| **Compliant Code** |
| --- |
| In this example the variable gets initialized and then it is able to be used freely as designed. |
| |  | | --- | | #include <iostream>    **void** f() {  **int** \*i = **new** **int**(12);  std::cout << i << ", " << \*i;  } | |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 7.0p0 | LANG.STRUCT.RPL  LANG.MEM.UVAR | Return pointer to local uninitialized variable |
| AStree | 20.10 | -Wuninitialized  Clang-analyzer-  Core.UndefinedBinaryOperatorResult | Does not catch all instances of this rule, such as uninitialed values read from heap-allocated memory |
| LDRA tool suite | 9.7.1 | 53 D, 69 D, 631 S, 652 S | Partially Implemented |
| Parasoft C\C++ test | 2022.1 | CERT\_CPP-EXP53-a | Avoid use before initialization |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-MEM50-CPP] | When memory is freed, all pointers into it become invalid, and its contents might either be returned to the operating system, making the freed space inaccessible, or remain intact and accessible. |

| **Noncompliant Code** |
| --- |
| In the following noncompliant code example, the dynamically allocated memory managed by the buff object is accessed after it has been implicitly deallocated by the object's destructor. |
| #include <iostream>  #include <memory>  #include <cstring>    **int** main(**int** argc, **const** **char** \*argv[]) {  **const** **char** \*s = "";  **if** (argc > 1) {  **enum** { BufferSize = 32 };  **try** {  std::unique\_ptr<**char**[]> buff(**new** **char**[BufferSize]);  std::**memset**(buff.get(), 0, BufferSize);  // ...  s = std::**strncpy**(buff.get(), argv[1], BufferSize - 1);  } **catch** (std::bad\_alloc &) {  // Handle error  }  }    std::cout << s << std::endl;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the lifetime of the buff object extends past the point at which the memory managed by the object is accessed. |
| |  | | --- | | #include <iostream>  #include <memory>  #include <cstring>    **int** main(**int** argc, **const** **char** \*argv[]) {  std::unique\_ptr<**char**[]> buff;  **const** **char** \*s = "";    **if** (argc > 1) {  **enum** { BufferSize = 32 };  **try** {  buff.reset(**new** **char**[BufferSize]);  std::**memset**(buff.get(), 0, BufferSize);  // ...  s = std::**strncpy**(buff.get(), argv[1], BufferSize - 1);  } **catch** (std::bad\_alloc &) {  // Handle error  }  }    std::cout << s << std::endl;  } | |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-  Cplusplus.NewDelete  clang-analyzer-  alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| CodeSonar | 7.0p0 | ALLOC.UAF | Use after free |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Helix QAC | 2022.2 | C++4303, C++4304 |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-CTR54-CPP] | When two pointers are subtracted, both must point to elements of the same array object or to one past the last element of the array object; the result is the difference of the subscripts of the two array elements. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, std::less<> is used in place of the < operator. The C++ Standard |
| |  | | --- | | #include <functional>  #include <iostream>    **template** <**typename** Ty>  **bool** in\_range(**const** Ty \*test, **const** Ty \*r, **size\_t** n) {  std::less<**const** Ty \*> less;  **return** !less(test, r) && less(test, r + n);  }    **void** f() {  **double** foo[10];  **double** \*x = &foo[0];  **double** bar;  std::cout << std::boolalpha << in\_range(&bar, x, 10);  } | |

| **Compliant Code** |
| --- |
| This compliant solution demonstrates a fully portable, but likely inefficient, implementation of in\_range() that compares test against each possible address in the range [r, n]. A compliant solution that is both efficient and fully portable is currently unknown. |
| #include <iostream>    **template** <**typename** Ty>  **bool** in\_range(**const** Ty \*test, **const** Ty \*r, **size\_t** n) {  auto \*cur = **reinterpret\_cast**<**const** unsigned **char** \*>(r);  auto \*end = **reinterpret\_cast**<**const** unsigned **char** \*>(r + n);  auto \*testPtr = **reinterpret\_cast**<**const** unsigned **char** \*>(test);    **for** (; cur != end; ++cur) {  **if** (cur == testPtr) {  **return** **true**;  }  }  **return** **false**;  }    **void** f() {  **double** foo[10];  **double** \*x = &foo[0];  **double** bar;  std::cout << std::boolalpha << in\_range(&bar, x, 10);  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.0p0 | LANG.STRUCT.CUP LANG.STRUCT.SUP | Comparison of Unrelated Pointers Subtraction of Unrelated Pointers |
| Helix QAC | 2022.2 | C++2668, C++2761, C++2762, C++2763, C++2766, C++2767, C++2768 |  |
| LDRA tool suite | 9.7.1 | 70 S, 87 S, 437 S, 438 S | Enhanced Enforcement |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-CTR54-a CERT\_CPP-CTR54-b CERT\_CPP-CTR54-c | Do not compare iterators from different containers Do not compare two unrelated pointers Do not subtract two pointers that do not address elements of the same array |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-ERR51-CPP] | When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. The C++ Standard |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| |  | | --- | | **void** throwing\_func() noexcept(**false**);    **void** f() {  throwing\_func();  }    **int** main() {  f();  } | |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| |  | | --- | | **void** throwing\_func() noexcept(**false**);    **void** f() {  throwing\_func();  }    **int** main() {  **try** {  f();  } **catch** (...) {  // Handle error  }  } | |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | MEdium | P4 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-ERR51-a CERT\_CPP-ERR51-b | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2022a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| CodeSonar | 7.0p0 | LANG.STRUCT.UCTCH | Unreachable Catch |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented | [STD-OOP50-CPP] | Virtual functions allow for the choice of member function calls to be determined at run time based on the dynamic type of the object that the member function is being called on. This convention supports object-oriented programming practices commonly associated with object inheritance and function overriding. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the base class attempts to seize and release an object's resources through calls to virtual functions from the constructor and destructor. However, the B::B() constructor calls B::seize() rather than D::seize(). Likewise, the B::~B() destructor calls B::release() rather than D::release(). |
| **struct** B {  B() { seize(); }  **virtual** ~B() { release(); }    **protected**:  **virtual** **void** seize();  **virtual** **void** release();  };    **struct** D : B {  **virtual** ~D() = **default**;    **protected**:  **void** seize() override {  B::seize();  // Get derived resources...  }    **void** release() override {  // Release derived resources...  B::release();  }  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the constructors and destructors call a nonvirtual, private member function (suffixed with mine) instead of calling a virtual function. The result is that each class is responsible for seizing and releasing its own resources |
| |  | | --- | | **class** B {  **void** seize\_mine();  **void** release\_mine();    **public**:  B() { seize\_mine(); }  **virtual** ~B() { release\_mine(); }    **protected**:  **virtual** **void** seize() { seize\_mine(); }  **virtual** **void** release() { release\_mine(); }  };    **class** D : **public** B {  **void** seize\_mine();  **void** release\_mine();    **public**:  D() { seize\_mine(); }  **virtual** ~D() { release\_mine(); }    **protected**:  **void** seize() override {  B::seize();  seize\_mine();  }    **void** release() override {  release\_mine();  B::release();  }  }; | |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-alpha.cplusplus.VirtualCall | Checked by clang-tidy |
| LDRA tool suite | 9.7.1 | 467 S, 92 D | Fully implemented |
| Helix QAC | 2022.2 | C++4260, C++4261, C++4273, C++4274, C++4275, C++4276, C++4277, C++4278, C++4279, C++4280, C++4281, C++4282 |  |
| CodeSonar | 7.0p0 | LANG.STRUCT.VCALL\_IN\_CTOR    LANG.STRUCT.VCALL\_IN\_DTOR | Virtual Call in Constructor    Virtual Call in Destructor |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Exclude user input | [STD-FIO30-C] | Never call a formatted I/O function with a format string containing a tainted value . An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location. |

| **Noncompliant Code** |
| --- |
| The incorrect\_password() function in this noncompliant code example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by user. This is an exemplar of untrusted data that originates from an unauthenticated user. The function constructs an error message that is then output to stderr using the C Standard fprintf() function. |
| |  | | --- | | #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **void** incorrect\_password(**const** **char** \*user) {  **int** ret;  /\* User names are restricted to 256 or fewer characters \*/  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + **sizeof**(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);  **if** (msg == NULL) {  /\* Handle error \*/  }  ret = snprintf(msg, len, msg\_format, user);  **if** (ret < 0) {  /\* Handle error \*/  } **else** **if** (ret >= len) {  /\* Handle truncated output \*/  }  **fprintf**(stderr, msg);  **free**(msg);  } | |

| **Compliant Code** |
| --- |
| This compliant solution fixes the problem by replacing the fprintf() call with a call to fputs(), which outputs msg directly to stderr without evaluating its contents: |
| |  | | --- | | #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **void** incorrect\_password(**const** **char** \*user) {  **int** ret;  /\* User names are restricted to 256 or fewer characters \*/  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + **sizeof**(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);  **if** (msg == NULL) {  /\* Handle error \*/  }  ret = snprintf(msg, len, msg\_format, user);  **if** (ret < 0) {  /\* Handle error \*/  } **else** **if** (ret >= len) {  /\* Handle truncated output \*/  }  **fputs**(msg, stderr);  **free**(msg);  } | |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO30 | Partially implemented |
| CodeSonar | 7.0p0 | IO.INJ.FMT MISC.FMT | Format string injection Format string |
| Coverity | 2017.07 | TAINTED\_STRING | Implemented |
| GCC | 4.3.5 |  | Can detect violations of this rule when the -Wformat-security flag is used |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Valid format string | [STD-FIO47-CPP] | The formatted output functions (fprintf() and related functions) convert, format, and print their arguments under control of a format string. |

| **Noncompliant Code** |
| --- |
| Mismatches between arguments and conversion specifications may result in undefined behavior. Compilers may diagnose type mismatches in formatted output function invocations. In this noncompliant code example, the error\_type argument to printf() is incorrectly matched with the s specifier rather than with the d specifier. |
| |  | | --- | | #include <stdio.h>    **void** func(**void**) {  **const** **char** \*error\_msg = "Resource not available to user.";  **int** error\_type = 3;  /\* ... \*/  **printf**("Error (type %s): %d\n", error\_type, error\_msg);  /\* ... \*/  } | |

| **Compliant Code** |
| --- |
| This compliant solution ensures that the arguments to the printf() function match their respective conversion specifications |
| #include <stdio.h>    **void** func(**void**) {  **const** **char** \*error\_msg = "Resource not available to user.";  **int** error\_type = 3;  /\* ... \*/  **printf**("Error (type %d): %s\n", error\_type, error\_msg);    /\* ... \*/  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | P6 | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO47 | Fully implemented |
| CodeSonar | 7.0p0 | IO.INJ.FMT  MISC.FMT  MISC.FMTTYPE | Format string injection  Format string  Format string type error |
| Coverity | 2017.07 | PW | Reports when the number of arguments differs from the number of required arguments according to the format string |
| GCC | 4.3.5 |  | Can detect violations of this recommendation when the -Wformat flag is used |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

[Insert your written explanations here.]

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Rule** | **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| **STD-001-CPP** | High | Unlikely | Medium | High | 2 |
| **[STD-DCL50-CPP]** | High | Unlikely | Medium | P12 | 1 |
| **[STD-DCL41-C]** | Medium | High | Medium | P4 | 3 |
| **[STD-STR50-CPP]** | High | High | Medium | P18 | 1 |
| **[STD-EXP53-CPP]** | High | High | Medium | P12 | 1 |
| **[STD-MEM50-CPP]** | High | High | Medium | P18 | 1 |
| **[STD-CTR54-CPP]** | Medium | Unlikely | Medium | P8 | 2 |
| **[STD-ERR51-CPP]** | Low | Likely | Medium | P4 | 3 |
| **[STD-OOP50-CPP]** | Low | Unlikely | Medium | P2 | 3 |
| **[STD-FIO30-C]** | High | High | Medium | P18 | 1 |
| **[STD-FIO47-CPP]** | High | Unlikely | Medium | P6 | 2 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | This keeps data secure when being stored. This means even when unactive, data still gets encrypted and secured. |
| Encryption at flight | This keeps data secure when being moved over a network connection. So when transferring data to someone or just moving it to another system using a network, the data will be secure and inaccessible to any other eyes. |
| Encryption in use | Encryption in use is where the data can be encrypted while being transferred, opened, shared, streamed, etc. This means there are a lot more capabilities of being able to show things off without worrying about others being able to see the unwanted. This keeps data very secure, especially now with how many things can be used live and be used in a public ongoign setting or within a group of people. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This is where users must authenticate their identity to access account information and be able to login or use the software provided. |
| Authorization | This is where users can give access to others or even to keep their own access more secure, potentially using 2-step verification or some means to lock others from attempting to log in. |
| Accounting | This would be data logs, trying to keep track of everything that happens on the account to see where errors happened or to just keep a log of the work done. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 08/12/2022 | Completed Worksheet | Michael Divis |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |